Python Radio 42: Buttons!

Simon Quellen Field

Simon Quellen Field

Follow

17 min read

·

Jun 8, 2025

Listen

Share

More

Reverse Engineering a Remote Control

Press enter or click to view image in full size

Remote control and its receiver.

All photos by the author.

There is a very nice little remote control available on AliExpress.com for three dollars. It comes with the receiver. Look for “433 MHz Wireless RF Remote Control Switch EV1527 Learning Code 4CH Relay Receiver Module and On Off Transmitter For DIY Kit”.

It is remarkably robust as a system, delivering long-range without false positives.

In the photo above, I connected four green LEDs to the outputs of the receiver. Pushing a button toggles the respective LED (D, C, B, or A). I pushed the B button to take the picture.

But what if we want to have a computer running Python send codes to the receiver? Or receive codes from the transmitter?

To do that, we would need to know the protocol the devices use to talk to one another.

Luckily, we have an RTL-SDR and some Python code to operate it.

Python Code for the RTL-SDR

Let’s look at the code I came up with:

Def normalize(data, limit):

L = len(data)

Result = [0] \* l

For i in range(l):

If abs(data[i]) >= limit:

Result[i] = 1

Return result

Def smooth(data):

Result = []

For i in range(len(data) – 11):

Result.append(sum(data[i:i+10])/10)

Return result

Def main():

From rtlsdr import RtlSdr

Import numpy as np

Import matplotlib.pyplot as plt

Sdr = RtlSdr()

Sdr.sample\_rate = 2048000 # Hz

Sdr.center\_freq = 433.92e6 # Hz

Sdr.freq\_correction = 60 # PPM

# print(sdr.valid\_gains\_db)

Sdr.gain = 49.6

# print(sdr.gain)

Sdr.read\_samples(4096) # Throw away the first few samples

Print(“Reading samples”)

X = sdr.read\_samples(2048000 \* 5)

Sdr.close()

Print(“Done reading samples”)

# Look for a 8,000 sample quiet period

Reals = x.real

Burst = []

For i in range(len(reals)):

For q in range(8\_000):

If abs(reals[i+q]) > .5:

I += q

Break

Else:

I += 100

While abs(reals[i]) < .5:

I += 1

Start = i – 10

Burst = reals[start:start+90\_000]

Plt.figure(figsize=(25, 2), dpi=100)

Plt.xlabel(“Milliseconds”)

Plt.xticks([x\*2048 for x in range(50)], [x for x in range(50)])

Plt.minorticks\_on()

Plt.plot(burst)

Plt.savefig(“rtlsdr.svg”, dpi=300)

Plt.tight\_layout()

Plt.show()

Burst = normalize(burst, .6)

Burst = smooth(burst)

Burst = normalize(burst, .4)

Plt.figure(figsize=(25, 2), dpi=100)

Plt.xlabel(“Milliseconds”)

Plt.xticks([x\*2048 for x in range(50)], [x for x in range(50)])

Plt.minorticks\_on()

Plt.plot(burst)

Plt.savefig(“rtlsdr\_canonical.svg”, dpi=300)

Longs = []

Shorts = []

I = 0

Stop = len(burst) – 5

While i < stop:

If burst[i]:

Count = 0

While i < stop and burst[i]:

Count += 1

I += 1

Print(f”{i}: {count} samples high {round(count / 2.048)} microseconds”)

If count > 100:

If count < 1000:

Shorts.append(count)

Else:

Longs.append(count)

Else:

Count = 0

While i < stop and burst[i] == 0:

Count += 1

I += 1

If count > 100:

If count < 3000:

If count < 1000:

Shorts.append(count)

Else:

Longs.append(count)

Print(f”{i}: {count} samples low {round(count / 2.048)} microseconds”)

If len(shorts) > 0 and len(longs) > 0:

Avg\_short = (sum(shorts) / len(shorts)) / 2.048

Avg\_long = (sum(longs) / len(longs)) / 2.048

Print(f”Average long pulse length: {avg\_long} microseconds”)

Print(f”Average short pulse length: {avg\_short} microseconds”)

Plt.tight\_layout()

Plt.show()

Exit()

Print(“No burst found”)

Plt.plot(reals)

Plt.legend([“Signal”, “Samples”])

Plt.savefig(“rtlsdr.svg”, dpi=300)

Plt.show()

Main()

At first, all I did was tell the RTL-SDR where to look (around 433.92 MHz) and gather some samples, showing them in a graph.

The Data

I got something that looked like this when I pushed the A button:

Press enter or click to view image in full size

Raw button data.

Some noise, then a short pulse, then some noise, then a long pulse, and so on. There were 25 pulses in all. The spacing between the pulses varied.

That’s when I wrote the normalise() and smooth() functions to clean up the data. Now it looked like this:

Press enter or click to view image in full size

Processed button data.

It looks like the short pulse is followed by a silence that is three times longer than the short pulse.

The long pulse is three times longer than the short pulse, and it is followed by a silence that is as long as the short pulse.

I decided to call short-long a zero, and long-short a one.

The four buttons produce these codes:

REMOTE\_BUTTON\_CODES = {

‘A’: “0110000011000011111110000”,

‘B’: “0110000011000011111101000”,

‘C’: “0110000011000011111100100”,

‘D’: “0110000011000011111100010”,

}

The last five bits are what distinguish the buttons. More likely, four bits and a stop bit. The first 20 bits are thus the address, or a sync code. This prevents noise from triggering the receiver.

The Timings

The RTL-SDR code prints out this information:

Found Rafael Micro R820T/2 tuner

Reading samples

Done reading samples

8: 8 samples low 4 microseconds

469: 461 samples high 225 microseconds

2529: 2060 samples low 1006 microseconds

4509: 1980 samples high 967 microseconds

5262: 753 samples low 368 microseconds

7277: 2015 samples high 984 microseconds

8039: 762 samples low 372 microseconds

8606: 567 samples high 277 microseconds

10694: 2088 samples low 1020 microseconds

11299: 605 samples high 295 microseconds

13392: 2093 samples low 1022 microseconds

13970: 578 samples high 282 microseconds

16051: 2081 samples low 1016 microseconds

16616: 565 samples high 276 microseconds

18722: 2106 samples low 1028 microseconds

19332: 610 samples high 298 microseconds

21403: 2071 samples low 1011 microseconds

23361: 1958 samples high 956 microseconds

23365: 4 samples low 2 microseconds

23381: 16 samples high 8 microseconds

23386: 5 samples low 2 microseconds

23395: 9 samples high 4 microseconds

24150: 755 samples low 369 microseconds

26147: 1997 samples high 975 microseconds

26924: 777 samples low 379 microseconds

27479: 555 samples high 271 microseconds

29577: 2098 samples low 1024 microseconds

30181: 604 samples high 295 microseconds

30183: 2 samples low 1 microseconds

30202: 19 samples high 9 microseconds

32288: 2086 samples low 1019 microseconds

32855: 567 samples high 277 microseconds

34939: 2084 samples low 1018 microseconds

35561: 622 samples high 304 microseconds

37656: 2095 samples low 1023 microseconds

39650: 1994 samples high 974 microseconds

39654: 4 samples low 2 microseconds

39682: 28 samples high 14 microseconds

40453: 771 samples low 376 microseconds

42460: 2007 samples high 980 microseconds

43230: 770 samples low 376 microseconds

45201: 1971 samples high 962 microseconds

45206: 5 samples low 2 microseconds

45223: 17 samples high 8 microseconds

46004: 781 samples low 381 microseconds

47994: 1990 samples high 972 microseconds

47995: 1 samples low 0 microseconds

48024: 29 samples high 14 microseconds

48790: 766 samples low 374 microseconds

50812: 2022 samples high 987 microseconds

51581: 769 samples low 375 microseconds

53579: 1998 samples high 976 microseconds

54361: 782 samples low 382 microseconds

56335: 1974 samples high 964 microseconds

56340: 5 samples low 2 microseconds

56355: 15 samples high 7 microseconds

57158: 803 samples low 392 microseconds

57687: 529 samples high 258 microseconds

57690: 3 samples low 1 microseconds

57706: 16 samples high 8 microseconds

59852: 2146 samples low 1048 microseconds

60463: 611 samples high 298 microseconds

62578: 2115 samples low 1033 microseconds

63142: 564 samples high 275 microseconds

63144: 2 samples low 1 microseconds

63161: 17 samples high 8 microseconds

65269: 2108 samples low 1029 microseconds

65801: 532 samples high 260 microseconds

65802: 1 samples low 0 microseconds

65820: 18 samples high 9 microseconds

75760: 9940 samples low 4854 microseconds

76223: 463 samples high 226 microseconds

78292: 2069 samples low 1010 microseconds

80798: 2506 samples high 1224 microseconds

81661: 863 samples low 421 microseconds

83706: 2045 samples high 999 microseconds

84451: 745 samples low 364 microseconds

84994: 543 samples high 265 microseconds

84997: 3 samples low 1 microseconds

85013: 16 samples high 8 microseconds

87106: 2093 samples low 1022 microseconds

87695: 589 samples high 288 microseconds

87700: 5 samples low 2 microseconds

87716: 16 samples high 8 microseconds

89791: 2075 samples low 1013 microseconds

89984: 193 samples high 94 microseconds

Average long pulse length: 1008.9742726293102 microseconds

Average short pulse length: 312.73626512096774 microseconds

This is the information we need to reverse engineer the device, using our CC1101 transceiver from the earlier projects (this one and this one).

The Micropython Code

Let’s build a program for the ESP32C3 Super Mini and the CC1101 to both sniff the data from the remote and send the four codes to the receiver:

Import gc

From machine import Pin, SPI, disable\_irq, enable\_irq

From time import sleep, sleep\_ms, sleep\_us, ticks\_diff, ticks\_ms, ticks\_us

From sys import print\_exception

From whoami import WhoAmI

BIT\_DECODE\_SHORT\_MIN = 200 # Min duration for a short pulse

BIT\_DECODE\_SHORT\_MAX = 700 # Max duration for a short pulse

BIT\_DECODE\_LONG\_MIN = 800 # Min duration for a long pulse

BIT\_DECODE\_LONG\_MAX = 1300 # Max duration for a long pulse

REMOTE\_BUTTON\_CODES = {

‘A’: “0110000011000011111110000”,

‘B’: “0110000011000011111101000”,

‘C’: “0110000011000011111100100”,

‘D’: “0110000011000011111100010”,

}

# Configuration Registers

IOCFG2 = 0x00; IOCFG1 = 0x01; IOCFG0 = 0x02; FIFOTHR = 0x03

SYNC1 = 0x04; SYNC0 = 0x05; PKTLEN = 0x06; PKTCTRL1 = 0x07

PKTCTRL0 = 0x08; ADDR = 0x09; CHANNR = 0x0A; FSCTRL1 = 0x0B

FSCTRL0 = 0x0C; FREQ2 = 0x0D; FREQ1 = 0x0E; FREQ0 = 0x0F

MDMCFG4 = 0x10; MDMCFG3 = 0x11; MDMCFG2 = 0x12; MDMCFG1 = 0x13

MDMCFG0 = 0x14; DEVIATN = 0x15; MCSM2 = 0x16; MCSM1 = 0x17

MCSM0 = 0x18; FOCCFG = 0x19; BSCFG = 0x1A; AGCCTRL2 = 0x1B

AGCCTRL1 = 0x1C; AGCCTRL0 = 0x1D; WOREVT1 = 0x1E; WOREVT0 = 0x1F

WORCTRL = 0x20; FREND1 = 0x21; FREND0 = 0x22; FSCAL3 = 0x23

FSCAL2 = 0x24; FSCAL1 = 0x25; FSCAL0 = 0x26; RCCTRL1 = 0x27

RCCTRL0 = 0x28; FSTEST = 0x29; PTEST = 0x2A; AGCTEST = 0x2B

TEST2 = 0x2C; TEST1 = 0x2D; TEST0 = 0x2E

# Status Registers

PARTNUM = 0xF0; VERSION = 0xF1; MARCSTATE = 0xF5; RSSI = 0xF4; LQI = 0xF3

TXBYTES = 0xFA; RXBYTES = 0xFB; PKTSTATUS = 0xF8

# Strobe Commands

SRES = 0x30; SRX = 0x34; STX = 0x35; SIDLE = 0x36; SCAL = 0x33

SFRX = 0x3A; SFTX = 0x3B; SNOP = 0x3D

# PATABLE and FIFO Addresses

PATABLE\_ADDR = 0x3E; TXFIFO\_ADDR = 0x3F; RXFIFO\_ADDR = 0x3F

# SPI Header Bits

WRITE\_SINGLE\_BYTE = 0x00; READ\_SINGLE\_BYTE = 0x80; WRITE\_BURST = 0x40; READ\_BURST = 0xC0

# States:

STATE\_SLEEP = 0x00

STATE\_IDLE = 0x01

STATE\_XOFF = 0x02

STATE\_VCOON\_MC = 0x03

STATE\_REGON\_MC = 0x04

STATE\_MANCAL = 0x05

STATE\_VCOON = 0x06

STATE\_REGON = 0x07

STATE\_STARTCAL = 0x08

STATE\_BWBOOST = 0x09

STATE\_FS\_LOCK = 0x0A

STATE\_IFADCON = 0x0B

STATE\_ENDCAL = 0x0C

STATE\_RX = 0x0D

STATE\_RX\_END = 0x0E

STATE\_RX\_RST = 0x0F

STATE\_TXRX\_SWITCH = 0x10

STATE\_RXFIFO\_OVERFLOW = 0x11

STATE\_FSTXON = 0x12

STATE\_TX = 0x13

STATE\_TX\_END = 0x14

STATE\_RXTX\_SWITCH = 0x15

STATE\_TXFIFO\_UNDERFLOW = 0x16

FXOSC = 26000000

Base\_frequency = 433.92

Channel = 0

SNIFF\_MAX\_FRAMES\_TO\_CAPTURE = 3

SNIFF\_MAX\_BITS\_PER\_FRAME = 24

SNIFF\_MIN\_PULSE\_US = 150

SNIFF\_MAX\_PULSE\_US = 2000

SNIFF\_MIN\_SYNC\_DURATION\_US = 3000

SNIFF\_FRAME\_TIMEOUT\_MS = 300

IDEAL\_T\_PULSE\_SHORT\_US = 350

IDEAL\_T\_PULSE\_LONG\_US = 1000

SYNC\_PULSE\_LOW\_US = 7000

Def accurate\_sleep\_us( delay ):

Irq\_state = disable\_irq()

Try:

Start\_wait = ticks\_us()

While ticks\_diff(ticks\_us(), start\_wait) < delay:

Pass

Finally:

Enable\_irq(irq\_state)

Import network

Import bluetooth

Def disable\_radios():

Sta\_if = network.WLAN(network.STA\_IF)

If sta\_if.active():

Sta\_if.active(False)

Print(“Wi-Fi STA disabled.”)

Ap\_if = network.WLAN(network.AP\_IF)

If ap\_if.active():

Ap\_if.active(False)

Print(“Wi-Fi AP disabled.”)

Try:

Ble = bluetooth.BLE()

If ble.active():

Ble.active(False)

Print(“Bluetooth disabled.”)

Except Exception as e:

Print(f”Could not disable Bluetooth (or not supported): {e}”)

Def enable\_radios():

Pass

Class CC1101\_ASK\_Tool:

Def \_\_init\_\_(self, spi, cs\_pin\_id, gdo0\_pin\_id):

Self.spi = spi

Self.cs = Pin(cs\_pin\_id, Pin.OUT)

Self.cs.on()

Self.current\_mode = None # “SNIFF\_RX” or “ASK\_TX”

If gdo0\_pin\_id is None:

Raise ValueError(“GDO0 pin is required for sniffing mode.”)

Self.data\_pin = Pin(gdo0\_pin\_id, Pin.IN, Pin.PULL\_DOWN)

Print(f”GDO0 (data input) configured on GPIO {gdo0\_pin\_id}”)

Self.reset()

Self.idle()

Print(“CC1101 ASK Tool Initialized and Idle.”)

Print(“--- Basic Register Read Test ---“)

Try:

Self.\_write\_reg(CHANNR, 0xBB)

Channr\_read = self.\_read\_reg(CHANNR)

If channr\_read == 0xBB:

Print(“Basic register write/read test PASSED.”)

Else:

Print(f”ERROR: Basic register write/read test FAILED! Wrote 0xBB to CHANNR, Read: 0x{channr\_read:02X}”)

Self.\_write\_reg(CHANNR, 0x00)

Except Exception as e:

Print\_exception(e)

Print(f”Error during basic register read test: {e}”)

Def \_strobe(self, cmd):

Self.cs.off(); self.spi.write(bytearray([cmd])); self.cs.on(); sleep\_us(50)

Def \_write\_reg(self, addr, value):

Self.cs.off(); self.spi.write(bytearray([addr | WRITE\_SINGLE\_BYTE, value])); self.cs.on(); sleep\_us(50)

Def \_read\_reg(self, addr):

Self.cs.off()

Wbuf = bytearray([addr | READ\_SINGLE\_BYTE, 0x00]); rbuf = bytearray(2)

Self.spi.write\_readinto(wbuf, rbuf); val = rbuf[1]

Self.cs.on(); sleep\_us(50); return val

Def \_read\_status\_reg(self, status\_addr\_with\_header):

Self.cs.off()

Wbuf = bytearray([status\_addr\_with\_header, 0x00]); rbuf = bytearray(2)

Self.spi.write\_readinto(wbuf, rbuf); val = rbuf[1]

Self.cs.on(); sleep\_us(50); return val

Def \_write\_burst\_reg(self, addr, data):

Self.cs.off(); self.spi.write(bytearray([addr | WRITE\_BURST])); self.spi.write(bytearray(data)); self.cs.on(); sleep\_us(50)

Def \_read\_burst\_reg(self, addr, length):

Self.cs.off()

Tx\_header\_byte = addr | READ\_BURST

Wbuf = bytearray([tx\_header\_byte] + [0x00] \* length)

Rbuf = bytearray(1 + length)

Self.spi.write\_readinto(wbuf, rbuf)

Data = rbuf[1:]

Self.cs.on(); sleep\_us(50); return data

Def reset(self):

Self.cs.off(); sleep\_us(10); self.cs.on(); sleep\_us(45)

Self.\_strobe(SRES); sleep\_ms(5)

Def idle(self):

Self.\_strobe(SIDLE)

Sleep\_ms(1)

For i in range(150):

Marc\_state = self.\_read\_status\_reg(MARCSTATE) & 0x1F

If marc\_state == 0x01: return

If i % 50 == 0 and i > 0 :

Sleep\_us(100)

Sleep\_us(50)

Print(f”Warning: CC1101 did not confirm IDLE. Last MARCSTATE: 0x{marc\_state:02X}”)

Def set\_frequency\_mhz(self, freq\_mhz=433.92, channel=0):

Freq\_hz = int(freq\_mhz \* 1\_000\_000 + channel \* 100\_000)

# print(f”Frequency is {freq\_hz}”)

Freq\_reg\_val = int((freq\_hz \* (1 << 16)) / FXOSC)

F2 = (freq\_reg\_val >> 16) & 0xFF

F1 = (freq\_reg\_val >> 8) & 0xFF

F0 = freq\_reg\_val & 0xFF

Self.\_write\_reg(FREQ2, f2); self.\_write\_reg(FREQ1, f1); self.\_write\_reg(FREQ0, f0)

Def configure\_for\_sniffer\_rx(self):

Self.reset()

Self.idle()

Self.set\_frequency\_mhz(base\_frequency, channel)

Self.\_write\_reg(IOCFG0, 0x0D)

Read\_iocfg0 = self.\_read\_reg(IOCFG0)

If read\_iocfg0 != 0x0D: print(f”ERROR: IOCFG0 not set to 0x0D, is 0x{read\_iocfg0:02X}”)

Self.\_write\_reg(MDMCFG2, 0x30)

Self.\_write\_reg(MDMCFG4, 0x6A)

Self.\_write\_reg(MDMCFG3, 0x22)

Self.\_write\_reg(DEVIATN, 0x00)

Self.\_write\_reg(PKTCTRL0,0x30)

Self.\_write\_reg(AGCCTRL2, 0x07)

Self.\_write\_reg(AGCCTRL1, 0x00)

Self.\_write\_reg(AGCCTRL0, 0xB0)

Self.\_write\_reg(FREND1, 0x56)

Self.\_write\_reg(MCSM0, 0x18)

Self.\_write\_reg(MCSM1, 0x0C)

Self.\_strobe(SFRX)

Self.\_strobe(SCAL)

Sleep\_ms(2)

Self.\_strobe(SRX)

Final\_marc\_state = 0x00

For attempt in range(10):

Current\_marc\_state = self.\_read\_status\_reg(MARCSTATE) & 0x1F

If current\_marc\_state == 0x0D:

Final\_marc\_state = current\_marc\_state

Break

Elif current\_marc\_state == 0x01:

Self.\_strobe(SRX)

Elif current\_marc\_state == 0x08:

Pass

Else:

Print(f”MARCSTATE is 0x{current\_marc\_state:02X} (unexpected) on check {attempt+1}.”)

Final\_marc\_state = current\_marc\_state

If attempt < 9:

Sleep\_ms(1)

Else:

Print(f”ERROR: MARCSTATE did not settle to RX (0x0D) after polling. Last state: 0x{final\_marc\_state:02X}.”)

Print(“Attempting full recovery sequence (IDLE, SFRX, SCAL, SRX)...”)

Self.idle()

Self.\_strobe(SFRX)

Self.\_strobe(SCAL)

Sleep\_ms(2)

Self.\_strobe(SRX)

Final\_marc\_state = self.\_read\_status\_reg(MARCSTATE) & 0x1F

If final\_marc\_state == 0x0D:

Sleep\_ms(100)

Self.current\_mode = “SNIFF\_RX”

Else:

Print(f”CRITICAL ERROR: CC1101 failed to enter RX mode. Final MARCSTATE: 0x{final\_marc\_state:02X}”)

Self.current\_mode = “ERROR\_SNIFF\_CONFIG”

Def configure\_for\_ask\_tx(self, pa\_table\_val=0xC0):

Print(“Configuring CC1101 for ASK/OOK Bit-Bang Transmission (PKTCTRL0=0x02)...”)

Self.reset()

Self.idle()

Self.set\_frequency\_mhz(base\_frequency, channel)

Self.\_write\_reg(PATABLE\_ADDR, pa\_table\_val)

Self.\_write\_reg(MDMCFG2, 0x30) # ASK/OOK

Self.\_write\_reg(DEVIATN, 0x00)

Self.\_write\_reg(PKTCTRL0, 0x02) # Synchronous serial mode. Packet handler off.

Self.\_write\_reg(PKTLEN, 1) # Set a dummy packet length. Required when PKTCTRL0[1:0] != 0b11.

Self.\_write\_reg(MDMCFG4, 0xA8)

Self.\_write\_reg(MDMCFG3, 0x93)

Self.\_write\_reg(FSCTRL1, 0x06)

Self.\_write\_reg(MCSM1, 0x01) # After TX (pulse): Go to FSTXON

Self.\_write\_reg(MCSM0, 0x09) # XOSC alw